14. Конструкции за цикъл

Проф. д-р Емил Хаджиколев

1. Цикъл
2. Видове цикли
3. Цикъл с пред-условие while
4. Цикъл със след-условие do-while
5. Цикъл с пред-условие for
6. Итератор
7. Основни циклични алгоритми
8. Вложени цикли

Цикъл (1)

- **Цикълът** (loop) **предоставя възможност за многократно изпълнение на** част от **кода**.
- Кодът, който се изпълнява многократно, се нарича **тяло на цикъла**.
- **Едно изпълнение на тялото на цикъл се нарича итерация.**
- **Изпълнението на цикъла зависи от** някакво **логическо условие**.
  - **Логическото условие е израз**, в който обикновено има променлива, чиято стойност се променя по време на итерациите.
  - **Параметър, от който зависи логическото условие, се нарича управляващ параметър.**
  - Може да има няколко управляващи параметри.

Цикъл (2)

- Преди да се изпълни една итерация, **се изчислява стойността на логическия израз**.
- **Ако той има стойност true**, тялото се изпълнява.
- **Когато логическото условие получи стойност false**, цикълът престава да се изпълнява.
- **За да не стане цикълът безкраен**, трябва условията да се задават правилно и внимателно да се управлява промяната на параметъра/параметрите.
- **В някои случаи нарочно се прави безкраен цикъл** и се използват операторите break и return за прекъсването му.

Примери за дейности, при които може да се ползват цикли

- **Обхождане на списък (напр. масив)**, изискващ еднотипна обработка на всеки отделен елемент – хора, фирми, числови (статистически) данни и много др.
- **Обработка на елементите на низ**, символ по символ.
- **Обхождане на елементите на матрица (таблица)** – чрез два цикъла –външният цикъл обхожда таблицата по редове, а вътрешния – по стълбове, и др.
- **Контрол на въвеждани от потребителя входни данни** – въвеждат се данни докато не станат коректни.

Видове цикли

- **Циклите са два основни вида: с пред-условие и със след-условие (пост-условие)**.
  - **При циклите с пред-условие**, условието се проверява преди изпълнението на итерацията.
  - **При циклите с пост-условие**, първо се изпълнява итерацията и след това се проверява условието (дали цикълът да приключи).
- За различните видове цикли, **се използват конструкциите** for, while, do-while.

Итератор

- **Итератор е механизъм за обхождане на колекции от елементи**, при който няма явно зададени условия.
- **Описва се с конструкцията for-each**.
- Чрез итератор може да се **обхождат всички елементи на колекция**.
  - По подразбиране елементите на подредени списъци (например масив) се обхождат от първия към последния (но реда може да се промени).
  - Реализира се чрез вътрешен (скрит) указател към следващ елемент.
  - В началото указателят реферира (сочи) първия елемент.
  - При всяка итерация се работи с елемента, сочен от указателя, а след изпълнението ѝ, указателят автоматично се пренасочва към следващия елемент.
  - Цикълът приключва, когато указателят спре да сочи следващ елемент.

Цикъл с пред-условие while

- **Синтаксисът на цикъла while** е:

```
while(<условие>){
  [<тяло>]
}
```

- **Може да се чете по следния начин**:

  Докато <условието> е истина, изпълнявай <тялото>!

- **Управляващият параметър** се декларира преди тялото на цикъла, а **стойността му се променя в тялото на цикъла**.
- При цикълът while е **възможно да не се изпълни нито една итерация**, ако условието още в началото има стойност false.

Пример (отпечатване на числата от 1 до 5)

```
int main()
{
                    // Желаем да отпечатаме числата от 1 до 5.
    int i = 1;      // На управляваща променлива i задаваме стойност 1.

    while (i <= 5) {          // Докато i е по-малко или равно на 5...
        cout << i << '\n';    // ...отпечатваме i...
        i++;                  // ...и увеличаваме i с единица.
    }

    return 0;
}
```

Цикъл със след-условие do-while

- **Синтаксисът на цикъла do-while** е:

```
do{
   [<тяло>]
} while(<условие>);
```

- **Може да се чете** по следния начин:

  Изпълнявай <тялото>, докато <условието> е истина!

- **Символът ; (точката и запетая) за край на конструкцията е задължителен**. Той не е задължителен за разгледания в предната част оператор за цикъл while.
- Както и при цикъла while, **управляващият параметър се декларира преди тялото на цикъла, а стойността му се променя в тялото** на цикъла.

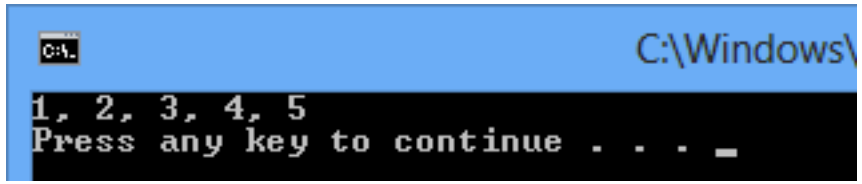
Пример (отпечатване на числата от 1 до 5)

```
int main()
{
    int i = 1;

    do {            // изпълнявай тялото...
        cout << i << (i<5 ? ", " : "");
        i++;
    } while (i <= 5); // ...докато i е по-малко или равно на 5

    cout << '\n';

    return 0;
}
```



Разлика между циклите с пред- и след-условие

- Разликата между цикъла със след-условие do-while и цикъла с пред-условие while е, че при do-while със сигурност ще се изпълни поне една итерация дори и условието да има стойност false.
- Нека например i има начална стойност 10. При do-while първо тялото ще се изпълни и ще се отпечата 10, след което ще се извърши за проверка на условието, което ще има стойност false…

Цикъл с пред-условие for (1)

- **Синтаксис на цикъла for**:

```
for([<инициализация на УП>];[<условие>];[<промяна на УП>]){
   [<тяло>]
}
```

- УП означава „управляващи променливи”.

Цикъл с пред-условие for (2)

- **Логиката при цикъла for е същата като при while**. Разликата е, че при for има заглавна (декларативна) част, в която има точно определени места за описание на основните дейности, свързани с управлението на цикъла:
- **Инициализация на управляващи променливи** – задават се начални стойности на управляващите променливи. Тази част се изпълнява еднократно в началото на изпълнението на конструкцията for.
- **Условие за изпълнение на тялото на цикъла –** изчислява се преди всяко изпълнение на тялото на цикъла. Ако условието е истина се изпълнява тялото на цикъла, в противен случай – изпълнението на цикъла приключва и започва изпълнението на следващата конструкция в програмата.
- **Промяна на управляващи променливи** – в тази част се променя стойността на управляващите променливи. Изпълнява се след всяко изпълнение на тялото на цикъла.

Цикъл с пред-условие for (3)

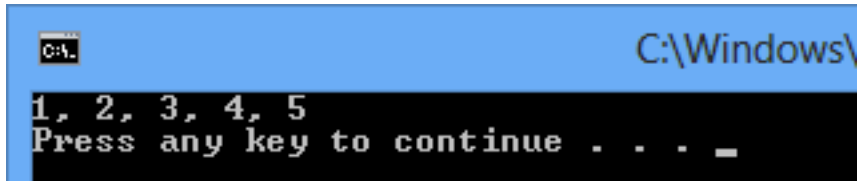
- **Трите части, разделени помежду си с ; (точка и запетая) не са задължителни**. Ако те не се опишат, цикълът става безкраен. В този случай може да се организира логиката за приключването му по друг начин – с командите break или return.
- **Чрез конструкцията за цикъл for се разделя логиката по управление на цикъла от работната логика.** Управлението е изнесено в декларативната част, а работната логика е в тялото на цикъла.
- **Това, че местата за основните дейности в конструкцията for са точно определени, ни предпазва от случайни грешки.** В показаните примери с while и do-while се променя стойността на управляващата променлива едва след като е изпълнена работната логиката – в случая, отпечатването на стойността на УП, каквато е и стандартната логика при for. Случайно или нарочно би могло да се извърши промяна на стойността ѝ на друго – неправилно място.

Пример (отпечатване на числата от 1 до 5)

```
int main(){
    for (int i = 1; i <= 5; i++) {
        cout << i << (i<5 ? ", " : "");
    }

    cout << '\n';

    return 0;
}
```



- **Забележка:**
  - В този пример, тъй като променливата i е декларирана в блока на цикъла for, тя ще е видима само в този блок.
  - Ако е необходимо да е достъпна и след изпълнението на цикъла, трябва да се декларира над него.

Няколко управляващи променливи - пример

- **В частта за инициализиране може да се задават стойности на няколко променливи**, като отделните присвоявания са разделени със запетаи. Аналогично, **в частта за промяна може да се променят стойностите на променливите**. Пример:

```
for (int i = 1, j = 20; i<j; i += 2, j -= 2) {
   cout << (j - i) << (i<j - 4 ? ", " : "");
}
```

Резултат:

```
19, 15, 11, 7, 3
```

- Тук, в частта за инициализация, за променливата i е зададена стойност 1, а за j – 20. След всяка итерация първата стойност се увеличава с две, а втората се намалява с две.

Извеждане на цели числа в указан интервал - пример

```
// Показва числата от n до m (n<m) в прав ред
void rightNumbers(int n, int m) {
    for (int i = n; i <= m; i++) {
        cout << i << (i < m ? ", " : "\n");
    }
}
// Показва числата от n до m (n<m) в обратен ред
void reverseNumbers(int n, int m) {
    for (int i = m; i >= n; i--) {
        cout << i << (i >n ? ", " : "\n");
    }
}
int main(){
    rightNumbers(4, 12);
    reverseNumbers(5, 9);
    return 0;
}
```

Резултат:

```
4, 5, 6, 7, 8, 9, 10, 11, 12
9, 8, 7, 6, 5
```

Извеждане на цели числа в указан интервал - разяснения

- В разгледания пример се извеждат целите числата в интервала [n, m], където n и m са параметри на функция.
- Създадени и използвани са две функции – за показване в прав и обратен ред.
- Особеност на функциите за показване на числата е, че първият параметър трябва да е по-малък от втория.
- По-добра реализация е ако параметрите са произволни числа.
- За целта, при необходимост първо се извършва размяна на стойностите на променливите:

```
void reverseNumbers(int n, int m) {
    if (n>m) {          // ако n>m - трябва да ги разменяме
        int temp = n;   // записваме n във временна променлива temp,
        n = m;          // Тогава на n задаваме m,
        m = temp;       // а на m - n чрез temp
    }

    for (int i = m; i >= n; i--) {
        cout << i << (i >n ? ", " : "\n");
    }
}
```

Сума на числа в указан интервал

- За намиране на сумата на целите числа в интервала [n, m] може да се използва цикъл (метода sum), в който се променя управляващата променлива i от n до m със стъпка 1.
- В началото сумата е 0, а при всяка итерация към нея се добавя стойността на i.

Сума на числа в указан интервал - пример

```
// Сума на целите числата от n до m (n<m)
void sum(int n, int m) {
    int sum = 0;                        // променлива за сума
    for (int i = n; i <= m; i++) {    // променяме i от n до m със стъпка 1
        sum += i;                      // при всяка итерация добавяме i към сумата
    }
    cout << "Сумата на числата от " << n << " до " << m << " е " << sum << '\n';
}
// Сума на елементите на аритметична прогресия
void sumArithmeticalProgression(int n, int m) {
    int sum = (m + n)*(m - n + 1) / 2; // (a0+an)*броя на елементите/2;
    cout << "Сумата на числата от " << n << " до " << m << " е " << sum << '\n';
}
int main(){
    sum(1, 5);
    sumArithmeticalProgression(1, 5);
    sum(5, 8);
    sumArithmeticalProgression(5, 8);
    return 0;
}
```

Резултат:

```
Сумата на числата от 1 до 5 е 15
Сумата на числата от 1 до 5 е 15
Сумата на числата от 5 до 8 е 26
Сумата на числата от 5 до 8 е 26
```

Сума на числа в указан интервал – разяснения (1)

- Както вече споменахме, „по-лесно” е да се намери сумата (функция sumArithmeticalProgression) като се отчете, че числата образуват аритметична прогресия с първи член n, последен – m, и брой на елементите (m-n+1).
- „По-лесно”, всъщност означава че алгоритъма за сума на аритметична прогресия е с константна сложност O(1), а сложността на цикъла е линейна: O(m-n), при която има (m-n+1)*3 основни действия (сравнение, увеличаване с 1 и сумиране).
- Ако интервала, за който искаме да извършим сумиране е например [1, 100000], при изпълнение на цикъла ще се реализират 100000 итерации, което ще е доста по-бавно от намирането на произведение на две суми и делението им.

Сума на числа в указан интервал – разяснения (2)

- Друга разлика в двата алгоритъма е, че при големи числа, втория – sumArithmeticalProgression по-бързо ще предизвиква препълване на типа int, който се използва за сума.
- Докато при цикъла само се добавя текущата стойност на i, във втория случай първо се изпълнява произведение на две големи числа и едва след това те се делят – (m+n)*(m-n+1)/2.
- За да се направи вторият алгоритъм по-добър при работа с големи числа, може да правим проверка кой от двата множителя (m+n) или (m-n+1) е четно число (единия със сигурност е четно число) и делението на 2 да се извърши с него и след това да се изчисли произведението с другия.
- Разбира се, и в двете решения може да се промени типа за сумата на long или long long.

Произведение на цели числа в указан интервал

- Произведението на последователни цели числа в указан интервал се реализира подобно на събирането – чрез цикъл. За него няма формула за бързо пресмятане.
- Тук особеност е, че **променливата за произведение първоначално се инициализира със стойност едно**. **При всяка итерация тя се умножава по текущата стойност на управляващата променлива**.
- В следващия пример е показано намирането на произведение на четните числа в интервала [n, m].
- В тялото на цикъла се обхождат всички числа от n до m. За всяко число се проверява дали е четно, и само ако е четно се умножава с намереното до текущата итерация произведение.

Произведение на четни числа в указан интервал - пример

```
// произведение на четни числа от n до m
void productEven(int n, int m) {
    long product = 1;
    for (int i = n; i <= m; i++) {  // променяме i от n до m със стъпка 1
        if (i % 2 == 0) {           // ако i се дели на 2...
                product *= i;       // го умножаваме с текущото произведение
        }
    }
    cout << "Произведението на четните числа от " << n <<  " до " << m << " е
    " << product << '\n';
}
int main(){
    productEven(1, 5);
    productEven(4, 9);
    return 0;
}
```

Резултат:

```
Произведението на четните числа от 1 до 5 е 8
Произведението на четните числа от 4 до 9 е 192
```

Произведение на четни числа в указан интервал – разяснения

- Може да се реализира и друга логика: стъпката на цикъла да е през две четни числа и въобще да няма проверка за четност в тялото му.
- За да стане това, достатъчно е първото число, с което се инициализира управляващата променлива, да е четно.
- В този случай сложността на алгоритъма ще си остане линейна, но ще се изпълнят двойно по-малко дейности. Новата реализация е показана в следващия код.

Произведение на четни числа в указан интервал – пример 2

```
void productEvenQuick(int n, int m) {
   long product = 1;
   if (n % 2 != 0) {  // ако n е нечетно...
      n++;            // ...става четно
   }
   for (int i = n; i <= m; i += 2) { // стъпка 2
      product *= i;
   }
   cout << "Произведението на четните числа от " << n <<  "
   до " << m << " е " << product << '\n';
}
```

Вложени цикли

- В тялото на един цикъл може да се описват различни други конструкции – if, switch, изрази, цикли и др.
- Вложени цикли има, когато **в тялото на един външен цикъл се изпълнява друг, вътрешен цикъл**. При всяка една итерация на външния цикъл, вътрешният изпълнява всичките си итерации.
- Например, ако външният цикъл има 10 итерации, а вътрешният 5, то тялото на вътрешния ще се изпълни общо 50 (=10*5) пъти.
- Чрез вложени цикли може да се реализират различни алгоритми за сортиране, да се обхождат матрици и др.

Умножение на числата от 1 до n

- Следващият пример представя таблицата за умножение на числата от 1 до n.
- Управляващата променлива на външния цикъл се променя от 1 до n.
- За всяка от стойностите ѝ по време на итерация, управляващата променлива на вътрешния цикъл също се променя от 1 до n.

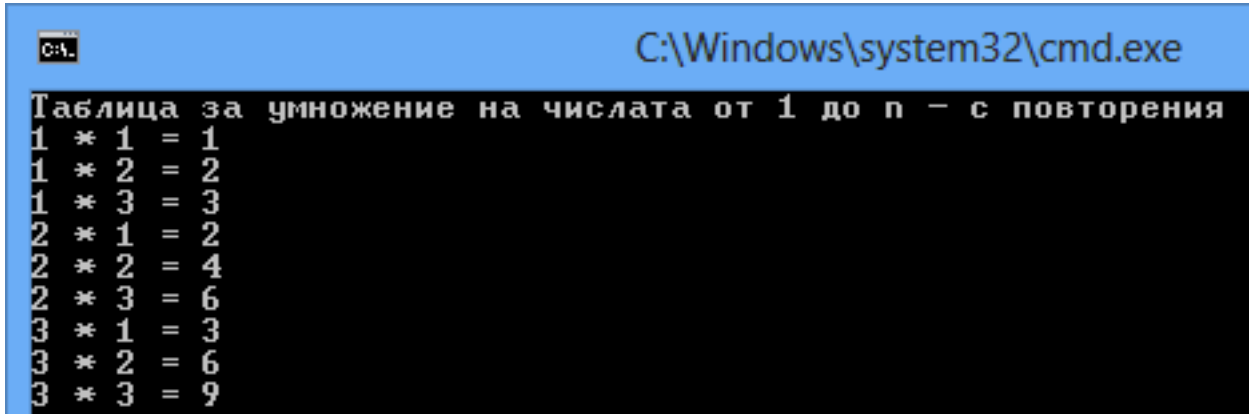
Умножение на числата от 1 до n - пример

```
void multiplicationTable(int n) {
    cout << "Таблица за умножение на числата от 1 до n - с повторения" << '\n';
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
                cout << i << " * " << j << " = " << i*j << '\n';
        }
    }
}

int main(){
    setlocale(LC_ALL, "bg");

    multiplicationTable(3);

    return 0;
}
```

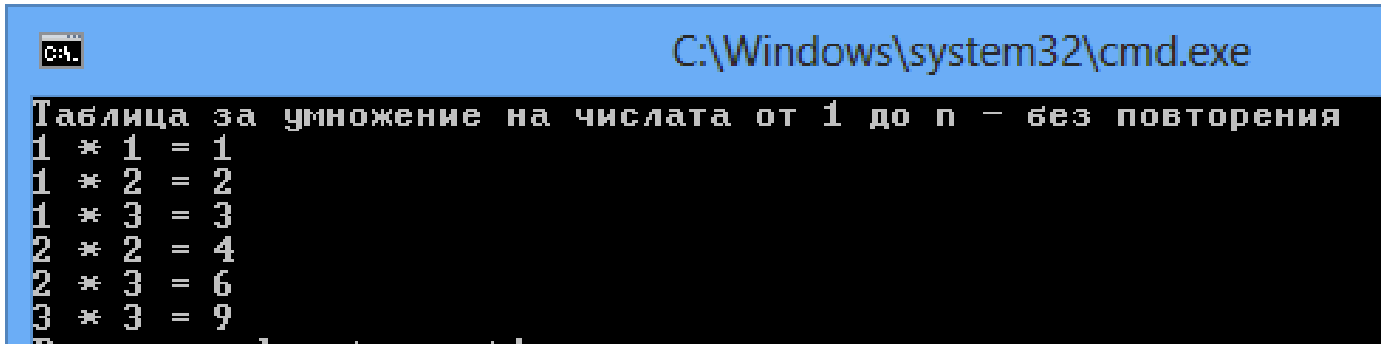


Умножение на числата от 1 до n – разяснения

- Както се вижда в резултата, някои от произведенията се повтарят, но с разменени места на множителите.
- **Повторенията може да се премахнат**. Повтарят се произведенията 2*1, 3*1, 3*2. Тази закономерност може да се обобщи: това са случаи, при които стойностите на j са по-малки от i. Следователно, решението е във вътрешния цикъл, j да се променя от i (а не от 1) до n.

Умножение на числата от 1 до n (4)

```
void multiplicationTableNoRepeats(int n) {
    cout << "Таблица за умножение на числата от 1 до n - без повторения"
    << '\n';
    for (int i = 1; i <= n; i++) {
        for (int j = i; j <= n; j++) {
                cout << i << " * " << j << " = " << i*j << '\n';
        }
    }
}
```



Ако функцията се стартира за n=3.